

# Lab information

Minikube provides a simple way to run applications in containers on Kubernetes.

## Lab overview

Lab 0: Provides a walkthrough for getting to know command-line tools and check if minikube is running.

Lab 1: This lab walks through creating and deploying a simple "guestbook" app written in Go as a net/http Server and accessing it.

Lab 2: Builds on lab 1 to expand to a more resilient setup which can survive having containers fail and recover. Lab 2 will also walk through basic services you need to get started with Kubernetes

Lab 3: Builds on lab 2 by increasing the capabilities of the deployed Guestbook application. This lab covers basic distributed application design and how kubernetes helps you use standard design practices.

Lab 4: How to enable your application so Kubernetes can automatically monitor and recover your applications with no user intervention.

Lab D: Debugging tips and tricks to help you along your Kubernetes journey. This lab is useful reference that does not follow in a specific sequence of the other labs.

## Download the Workshop

# Source Code

Repo `guestbook` has the application that we'll be deploying. While we're not going to build it we will use the deployment configuration files from that repo. Guestbook application has two versions v1 and v2 which we will use to demonstrate some rollout functionality later. All the configuration files we use are under the directory `guestbook/v1`.

```
$ git clone https://github.com/niklaushirt/guestbook.git
```

## Minikube

### Make sure minikube is running

Verify that minikube is running If not please complete KUB01 Lab Setup

```
$ minikube status
```

```
host: Running
kubelet: Running
apiserver: Running
kubectl: Correctly Configured: pointing to minikube-vm at 192.
```



Verify kubectl can communicate with your cluster.

```
$ kubectl get nodes
```

NAME	STATUS	ROLES	AGE	VERSION
minikube	Ready	master	32m	v1.14.1

### Enable Kubernetes Dashboard

```
$ minikube dashboard
```

```
🔔 Enabling dashboard ...  
🤔 Verifying dashboard health ...  
🚀 Launching proxy ...  
🤔 Verifying proxy health ...  
🎉 Opening http://127.0.0.1:58935/api/v1/namespaces/kube-system/s
```

After a while the Kubernetes Dashboard will open

## Getting to know minikube

### What is Minikube?

Minikube is an open source tool that enables you to run Kubernetes on your laptop or other local machine. It can work with Linux, Mac, and Windows operating systems. It runs a single-node cluster inside a virtual machine on your local machine.

minikube runs the official stable release of Kubernetes, with support for standard Kubernetes features like:

- LoadBalancer - using `minikube tunnel`

- Multi-cluster - using `minikube start -p <name>`

- NodePorts - using `minikube service`

- Persistent Volumes

- Ingress

- RBAC

- Dashboard - `minikube dashboard`

- Container runtimes

- Configure apiserver and kubelet options via command-line flags

- Addons - a marketplace for developers to share configurations for running services on minikube

## Useful commands

Start a cluster by running:

```
minikube start
```

Access Kubernetes Dashboard within Minikube:

```
minikube dashboard
```

Open this exposed endpoint in your browser:

```
minikube service hello-minikube
```

Start a second local cluster:

```
minikube start -p cluster2
```

Stop your local cluster:

```
minikube stop
```

Delete your local cluster:

```
minikube delete
```

## Lab 0. Get to know kubectl

Learn how to use the `kubectl` command line.

Open your terminal:

On your Mac you can hit Command + Space and type Terminal.

Type `kubectl` and you will get the list of all available commands.

```
$ kubectl
```

```
kubectl controls the Kubernetes cluster manager.
```

Find more information at: <https://kubernetes.io/docs/reference/kub>

#### Basic Commands (Beginner):

<code>create</code>	Create a resource from a file or from stdin.
<code>expose</code>	Take a replication controller, service, deployment
<code>run</code>	Run a particular image on the cluster
<code>set</code>	Set specific features on objects

#### Basic Commands (Intermediate):

<code>explain</code>	Documentation of resources
<code>get</code>	Display one or many resources
<code>edit</code>	Edit a resource on the server
<code>delete</code>	Delete resources by filenames, stdin, resources a
...	



They allow you to work with Kubernetes objects.

The most important ones are:

**`create`** Create a resource from a file or from stdin.

**`delete`** Delete resources by filenames, stdin, resources and names, or by resources

**`get`** Display one or many resources

**`describe`** Show details of a specific resource or group of resources

For example you can get the nodes in your cluster by typing

```
kubectl get nodes
```

## Lab 0. Get to know yaml

YAML ("YAML Ain't Markup Language") is a human-readable data serialization language. It is commonly used for configuration files, but could be used in many applications where data is being stored (e.g. debugging output) or transmitted (e.g. document headers).

The YAML format is generally used to describe Kubernetes objects.

Whitespace indentation is used to denote structure; however, tab characters are never allowed as indentation.

Comments begin with the number sign (#)

List members are denoted by a leading hyphen (-) with one member per line

Strings are ordinarily unquoted, but may be enclosed in double-quotes ("), or single-quotes (').

## **guestbook-deployment.yaml**

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: guestbook-v1
  labels:
    app: guestbook
    version: "1.0"
spec:
  replicas: 3
  selector:
    matchLabels:
      app: guestbook
  template:
    metadata:
      labels:
        app: guestbook
        version: "1.0"
    spec:
      containers:
        - name: guestbook
          image: ibmcom/guestbook:v1
          ports:
            - name: http-server
              containerPort: 3000
```

In this example, the structure `metadata` contains `name` and `labels`. `labels` in turn contains `app` and `version`.

# **Lab 1. Set up and deploy your**

# first application

Learn how to deploy an application to a Kubernetes cluster.

Once your client is configured, you are ready to deploy your first application, `guestbook`.

## 1. Deploy your application

In this part of the lab we will deploy an application called `guestbook` that has already been built and uploaded to DockerHub under the name `ibmcom/guestbook:v1`.

### 1. Start by running `guestbook`

```
$ kubectl run guestbook --image=ibmcom/guestbook:v1
```

This action will take a bit of time. To check the status of the running application, you can use `kubectl get pods`.

You should see output similar to the following:

```
$ kubectl get pods
```

NAME	READY	STATUS	RESTARTS
guestbook-59bd679fdc-bxdg7	0/1	ContainerCreating	0



Eventually, the status should show up as **Running**.

```
$ kubectl get pods
```

NAME	READY	STATUS	RESTARTS
guestbook-59bd679fdc-bxdg7	1/1	Running	0



The end result of the `run` command is not just the pod containing our application containers, but a Deployment

resource that manages the lifecycle of those pods.

2. Once the status reads **Running**, we need to expose that deployment as a service so we can access it through the IP of the worker nodes. The **guestbook** application listens on port 3000. Run:

```
$ kubectl expose deployment guestbook --type="NodePort" --port=3000
service "guestbook" exposed
```

3. To find the port used on that worker node, examine your new service:

```
$ kubectl get service guestbook
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)
guestbook	NodePort	10.10.10.253	<none>	3000:31208

We can see that our **<nodeport>** is **31208**. We can see in the output the port mapping from 3000 inside the pod exposed to the cluster on port 31208. This port in the 31000 range is automatically chosen, and could be different for you.

4. **guestbook** is now running on your cluster, and exposed to the internet. We need to find out where it is accessible. The worker nodes running in the container service get external IP addresses. Run **minikube status**, and note the public IP listed on the **pointing to minikube-vm** at **www.xxx.yyy.zzz** line.

```
$ minikube status
```

```
host: Running
kubelet: Running
apiserver: Running
kubectl: Correctly Configured: pointing to minikube-vm at 192.168.
```



We can see that our `<public-IP>` is `192.168.99.100`.

1. Now that you have both the address and the port, you can now access the application in the web browser at `<public-IP>:<nodeport>`. In the example case this is `192.168.99.100:31208`.

## Hint

For your convenience, you can open the webpage directly by typing

```
minikube service guestbook
```

where `guestbook` is the name of the exposed kubernetes service.

Congratulations, you've now deployed an application to Kubernetes!

We will be using this deployment in the next lab of this course (Lab2).

You should now go back up to the root of the repository in preparation for the next lab:

```
cd ..
```

# Lab 2: Scale and Update Deployments

In this lab, you'll learn how to update the number of instances a deployment has and how to safely roll out an update of your application on Kubernetes.

For this lab, you need a running deployment of the `guestbook` application from the previous lab. If you deleted it, recreate it using:

```
$ kubectl run guestbook --image=ibmcom/guestbook:v1
```

## Scale apps with replicas

A *replica* is a copy of a pod that contains a running service. By having multiple replicas of a pod, you can ensure your deployment has the available resources to handle increasing load on your application.

1. **kubectl** provides a **scale** subcommand to change the size of an existing deployment. Let's increase our capacity from a single running instance of **guestbook** up to 10 instances:

```
$ kubectl scale --replicas=10 deployment guestbook
deployment "guestbook" scaled
```

Kubernetes will now try to make reality match the desired state of 10 replicas by starting 9 new pods with the same configuration as the first.

2. To see your changes being rolled out, you can run: **kubectl rollout status deployment guestbook**.

The rollout might occur so quickly that the following messages might *not* display:

```
$ kubectl rollout status deployment guestbook
```

```
Waiting for rollout to finish: 1 of 10 updated replicas are available
Waiting for rollout to finish: 2 of 10 updated replicas are available
Waiting for rollout to finish: 3 of 10 updated replicas are available
Waiting for rollout to finish: 4 of 10 updated replicas are available
Waiting for rollout to finish: 5 of 10 updated replicas are available
Waiting for rollout to finish: 6 of 10 updated replicas are available
Waiting for rollout to finish: 7 of 10 updated replicas are available
Waiting for rollout to finish: 8 of 10 updated replicas are available
Waiting for rollout to finish: 9 of 10 updated replicas are available
deployment "guestbook" successfully rolled out
```



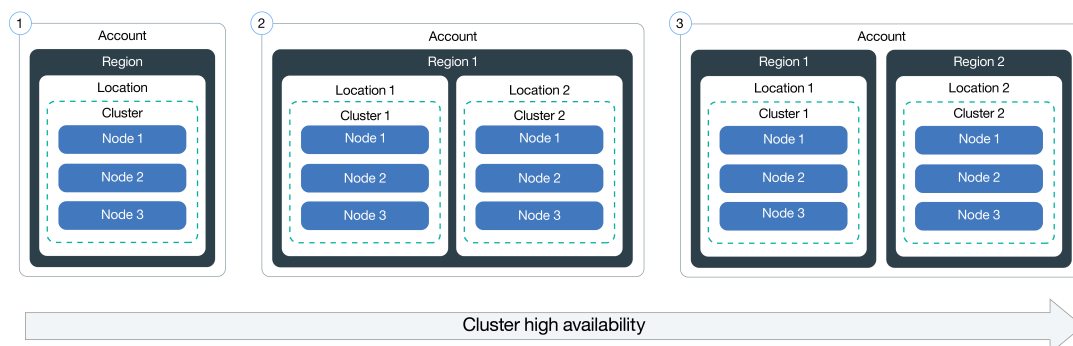
3. Once the rollout has finished, ensure your pods are running by using: `kubectl get pods`.

You should see output listing 10 replicas of your deployment:

```
$ kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
guestbook-562211614-1tqm7	1/1	Running	0	1d
guestbook-562211614-1zqn4	1/1	Running	0	2m
guestbook-562211614-5htdz	1/1	Running	0	2m
guestbook-562211614-6h04h	1/1	Running	0	2m
guestbook-562211614-ds9hb	1/1	Running	0	2m
guestbook-562211614-nb5qp	1/1	Running	0	2m
guestbook-562211614-vtfp2	1/1	Running	0	2m
guestbook-562211614-vz5qw	1/1	Running	0	2m
guestbook-562211614-zksw3	1/1	Running	0	2m
guestbook-562211614-zsp0j	1/1	Running	0	2m

**Tip:** Another way to improve availability is to add clusters and regions to your deployment, as shown in the following diagram:



## Update and roll back apps

Kubernetes allows you to do rolling upgrade of your application to a new container image. This allows you to easily update the running image and also allows you to easily undo a rollout if a problem is discovered during or after deployment.

In the previous lab, we used an image with a `v1` tag. For our upgrade we'll use the image with the `v2` tag.

To update and roll back: 1. Using `kubectl`, you can now update your deployment to use the `v2` image. `kubectl` allows you to change details about existing resources with the `set` subcommand. We can use it to change the image being used.

```
kubectl set image deployment/guestbook
guestbook=ibmcom/guestbook:v2
```

Note that a pod could have multiple containers, each with its own name. Each image can be changed individually or all at once by referring to the name. In the case of our `guestbook` Deployment, the container name is also `guestbook`. Multiple containers can be updated at the same time. ([More information.](#))

### 1. Run `kubectl rollout status`

`deployment/guestbook` to check the status of the rollout. The rollout might occur so quickly that the following messages might *not* display:

```
$ kubectl rollout status deployment/guestbook
```

```
Waiting for rollout to finish: 2 out of 10 new replicas have t
Waiting for rollout to finish: 3 out of 10 new replicas have t
Waiting for rollout to finish: 3 out of 10 new replicas have t
Waiting for rollout to finish: 3 out of 10 new replicas have t
Waiting for rollout to finish: 4 out of 10 new replicas have t
Waiting for rollout to finish: 4 out of 10 new replicas have t
Waiting for rollout to finish: 4 out of 10 new replicas have t
Waiting for rollout to finish: 4 out of 10 new replicas have t
Waiting for rollout to finish: 4 out of 10 new replicas have t
Waiting for rollout to finish: 5 out of 10 new replicas have t
Waiting for rollout to finish: 5 out of 10 new replicas have t
Waiting for rollout to finish: 5 out of 10 new replicas have t
Waiting for rollout to finish: 6 out of 10 new replicas have t
Waiting for rollout to finish: 6 out of 10 new replicas have t
Waiting for rollout to finish: 6 out of 10 new replicas have t
Waiting for rollout to finish: 7 out of 10 new replicas have t
Waiting for rollout to finish: 7 out of 10 new replicas have t
Waiting for rollout to finish: 7 out of 10 new replicas have t
Waiting for rollout to finish: 8 out of 10 new replicas have t
Waiting for rollout to finish: 8 out of 10 new replicas have t
Waiting for rollout to finish: 8 out of 10 new replicas have t
```

```
Waiting for rollout to finish: 8 out of 10 new replicas have t
Waiting for rollout to finish: 9 out of 10 new replicas have t
Waiting for rollout to finish: 9 out of 10 new replicas have t
Waiting for rollout to finish: 9 out of 10 new replicas have t
Waiting for rollout to finish: 1 old replicas are pending terr
Waiting for rollout to finish: 1 old replicas are pending terr
Waiting for rollout to finish: 1 old replicas are pending terr
Waiting for rollout to finish: 9 of 10 updated replicas are a
Waiting for rollout to finish: 9 of 10 updated replicas are a
Waiting for rollout to finish: 9 of 10 updated replicas are a
deployment "guestbook" successfully rolled out
```



2. Test the application as before, by accessing `<public-IP>:<nodeport>` in the browser to confirm your new code is active.

Remember, to get the "nodeport" and "public-ip" use:

```
kubectl describe service guestbook and
minikube status
```

### Hint

For your convenience, you can open the webpage directly by typing

```
$ minikube service guestbook
```

where guestbook is the name of the exposed kubernetes service.

To verify that you're running "v2" of guestbook, look at the title of the page, it should now be **Guestbook - v2**

Hint If the page doesn't show the V2 label reload the webpage without caching - usually done by holding SHIFT and reload. Otherwise empty the browser cache.

3. If you want to undo your latest rollout, use: `$ kubectl rollout undo deployment guestbook deployment`

"guestbook"

You can then use `kubectl rollout status deployment/guestbook` to see the status.

4. When doing a rollout, you see references to *old* replicas and *new* replicas. The *old* replicas are the original 10 pods deployed when we scaled the application. The *new* replicas come from the newly created pods with the different image. All of these pods are owned by the Deployment. The deployment manages these two sets of pods with a resource called a ReplicaSet. We can see the guestbook ReplicaSets with:

```
$ kubectl get replicaset -l run=guestbook
```

NAME	DESIRED	CURRENT	READY	AGE
guestbook-5f5548d4f	10	10	10	21m
guestbook-768cc55c78	0	0	0	3h

Before we continue, let's delete the application so we can learn about a different way to achieve the same results:

To remove the deployment, use `kubectl delete deployment guestbook`.

To remove the service, use `kubectl delete service guestbook`.

Congratulations! You deployed the second version of the app. Lab 2 is now complete.

## Lab 3: Scale and update apps natively, building multi-tier applications.

In this lab you'll learn how to deploy the same guestbook

application we deployed in the previous labs, however, instead of using the `kubectl` command line helper functions we'll be deploying the application using configuration files. The configuration file mechanism allows you to have more fine-grained control over all of resources being created within the Kubernetes cluster.

You have already cloned the github repo:

```
$ git clone https://github.com/niklaushirt/guestbook.git
```

This repo contains multiple versions of the guestbook application as well as the configuration files we'll use to deploy the pieces of the application.

Change directory by running the command `cd guestbook`. You will find all the configurations files for this exercise under the directory `v1`.

## Scale apps natively

Kubernetes can deploy an individual pod to run an application but when you need to scale it to handle a large number of requests a **Deployment** is the resource you want to use. A Deployment manages a collection of similar pods. When you ask for a specific number of replicas the Kubernetes Deployment Controller will attempt to maintain that number of replicas at all times.

Every Kubernetes object we create should provide two nested object fields that govern the object's configuration: the object **spec** and the object **status**. Object **spec** defines the desired state, and object **status** contains Kubernetes system provided information about the actual state of the resource. As described before, Kubernetes will attempt to reconcile your desired state with the actual state of the system.

For Object that we create we need to provide the **apiVersion** you are using to create the object, **kind** of the object we are

creating and the **metadata** about the object such as a **name**, set of **labels** and optionally **namespace** that this object should belong.

Consider the following deployment configuration for guestbook application (you find those in the v1 directory of the cloned lab source code):

### **guestbook-deployment.yaml**

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: guestbook
  labels:
    app: guestbook
spec:
  replicas: 3
  selector:
    matchLabels:
      app: guestbook
  template:
    metadata:
      labels:
        app: guestbook
    spec:
      containers:
        - name: guestbook
          image: ibmcom/guestbook:v1
          ports:
            - name: http-server
              containerPort: 3000
```

The above configuration file create a deployment object named 'guestbook' with a pod containing a single container running the image **ibmcom/guestbook:v1**. Also the configuration specifies replicas set to 3 and Kubernetes tries to make sure that at least three active pods are running at all times.

Create guestbook deployment

To create a Deployment using this configuration file we use



the following command:

```
$ cd guestbook/v1/  
$ kubectl create -f guestbook-deployment.yaml  
  
deployment "guestbook" created
```

List the pod with label app=guestbook

We can then list the pods it created by listing all pods that have a label of "app" with a value of "guestbook". This matches the labels defined above in the yaml file in the `spec.template.metadata.labels` section.

```
$ kubectl get pods -l app=guestbook
```

When you change the number of replicas in the configuration, Kubernetes will try to add, or remove, pods from the system to match your request.

Open the deployment file we used to create the Deployment in your preferred editor to make changes. You'll notice that there are a lot more fields in this version than the original yaml file we used. This is because it contains all of the properties about the Deployment that Kubernetes knows about, not just the ones we chose to specify when we create it.

You should use the following command to make the change effective when you have finished editing the yaml file.

```
$ kubectl apply -f guestbook-deployment.yaml
```

This will ask Kubernetes to "diff" our yaml file with the current state of the Deployment and apply just those changes.

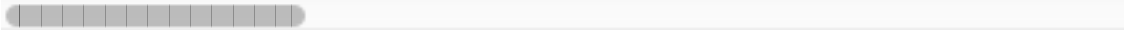
## Hint

If there is a deployment error please check that your kubectl

version is the same as the kubernetes version running in minikube. The `client version` should match the `server version`:

```
$ kubectl version
```

```
Client Version: version.Info{Major:"1", Minor:"14", GitVersion:"v1.14.0", GitCommit:"6438c0255b7701575073a1470d091076627443ee",
Server Version: version.Info{Major:"1", Minor:"14", GitVersion:"v1.14.0", GitCommit:"6438c0255b7701575073a1470d091076627443ee",
```



We can now define a Service object to expose the deployment to external clients.

### **guestbook-service.yaml**

```
apiVersion: v1
kind: Service
metadata:
  name: guestbook
  labels:
    app: guestbook
spec:
  ports:
    - port: 3000
      targetPort: http-server
  selector:
    app: guestbook
  type: LoadBalancer
```

The above configuration creates a Service resource named `guestbook`. A Service can be used to create a network path for incoming traffic to your running application. In this case, we are setting up a route from port 3000 on the cluster to the "http-server" port on our app, which is port 3000 per the Deployment container spec.

Let us now create the `guestbook` service using the same type of command we used when we created the Deployment:

```
$ kubectl create -f guestbook-service.yaml
```

Test guestbook app using a browser of your choice using the url `<your-cluster-ip>:<node-port>`

Remember, to get the `nodeport` and `public-ip` use:

```
$ kubectl describe service guestbook
```

and

```
$ minikube status
```

## Hint

For your convenience, you can open the webpage directly by typing

```
minikube service guestbook
```

where `guestbook` is the name of the exposed kubernetes service.

## Connect to a back-end service

If you look at the guestbook source code, under the `guestbook/v1/guestbook` directory, you'll notice that it is written to support a variety of data stores. By default it will keep the log of guestbook entries in memory. That's ok for testing purposes, but as you get into a more "real" environment where you scale your application that model will not work because based on which instance of the application the user is routed to they'll see very different results.

To solve this we need to have all instances of our app share the same data store - in this case we're going to use a redis database that we deploy to our cluster. This instance of redis will be defined in a similar manner to the guestbook.

# Deploy Redis Master

## redis-master-deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: redis-master
  labels:
    app: redis
    role: master
spec:
  replicas: 1
  selector:
    matchLabels:
      app: redis
      role: master
  template:
    metadata:
      labels:
        app: redis
        role: master
    spec:
      containers:
        - name: redis-master
          image: redis:2.8.23
          ports:
            - name: redis-server
              containerPort: 6379
```

This yaml creates a redis database in a Deployment named 'redis-master'. It will create a single instance, with replicas set to 1, and the guestbook app instances will connect to it to persist data, as well as read the persisted data back. The image running in the container is 'redis:2.8.23' and exposes the standard redis port 6379.

Create a redis Deployment, like we did for guestbook:

```
$ kubectl create -f redis-master-deployment.yaml
```

Check to see that redis server pod is running:

```
$ kubectl get pods -l app=redis,role=master
NAME                READY    STATUS    RESTARTS   AGE
redis-master-q9zg7  1/1     Running   0           2d
```

Let us test the redis standalone:

```
$ kubectl exec -it redis-master-q9zg7 redis-
cli
```

The `kubectl exec` command will start a secondary process in the specified container. In this case we're asking for the "redis-cli" command to be executed in the container named "redis-master-q9zg7". When this process ends the "kubectl exec" command will also exit but the other processes in the container will not be impacted.

Once in the container we can use the "redis-cli" command to make sure the redis database is running properly, or to configure it if needed.

```
redis-cli> ping
PONG
redis-cli> exit
```

## Expose Redis Master

Now we need to expose the `redis-master` Deployment as a Service so that the guestbook application can connect to it through DNS lookup.

### redis-master-service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: redis-master
labels:
  app: redis
  role: master
```

```
spec:
  ports:
  - port: 6379
    targetPort: redis-server
  selector:
    app: redis
    role: master
```

This creates a Service object named 'redis-master' and configures it to target port 6379 on the pods selected by the selectors "app=redis" and "role=master".

Create the service to access redis master:

```
$ kubectl create -f redis-master-service.yaml
```

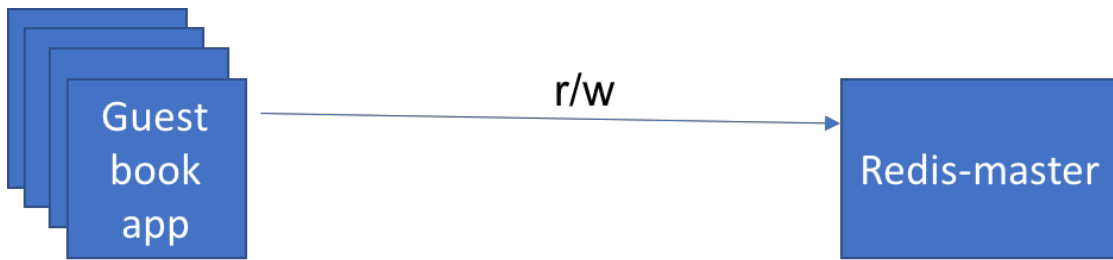
Restart guestbook so that it will find the redis service to use database:

```
$ kubectl delete deploy guestbook-v1
$ kubectl create -f guestbook-deployment.yaml
```

Test guestbook app using a browser of your choice using the url: **<your-cluster-ip>:<node-port>**

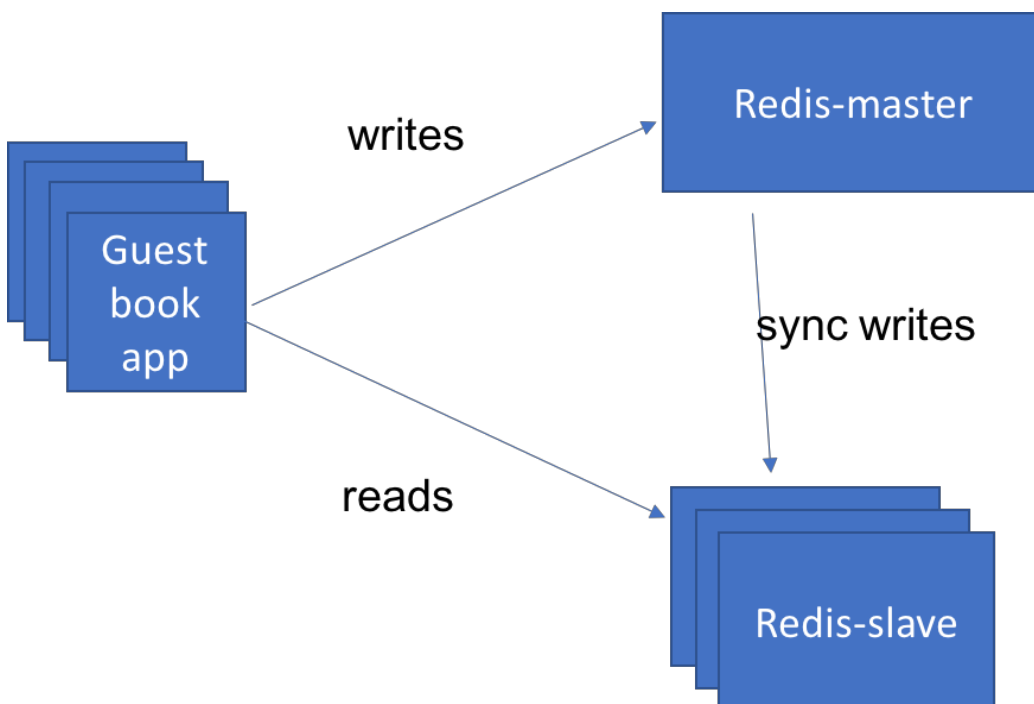
You can see now that if you open up multiple browsers and refresh the page to access the different copies of guestbook that they all have a consistent state. All instances write to the same backing persistent storage, and all instances read from that storage to display the guestbook entries that have been stored.

We have our simple 3-tier application running but we need to scale the application if traffic increases. Our main bottleneck is that we only have one database server to process each request coming through guestbook. One simple solution is to separate the reads and write such that they go to different databases that are replicated properly to achieve data consistency.



## Deploy Redis Slave

Create a deployment named 'redis-slave' that can talk to redis database to manage data reads. In order to scale the database we use the pattern where we can scale the reads using redis slave deployment which can run several instances to read. Redis slave deployments is configured to run two replicas.



### redis-slave-deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: redis-slave
  labels:
```

```

    app: redis
    role: slave
spec:
  replicas: 2
  selector:
    matchLabels:
      app: redis
      role: slave
  template:
    metadata:
      labels:
        app: redis
        role: slave
    spec:
      containers:
        - name: redis-slave
          image: kubernetes/redis-slave:v2
          ports:
            - name: redis-server
              containerPort: 6379

```

Create the pod running redis slave deployment. `$ kubectl create -f redis-slave-deployment.yaml`

Check if all the slave replicas are running

```

$ kubectl get pods -l app=redis,role=slave
NAME                READY    STATUS    RESTARTS   AGE
redis-slave-kd7vx   1/1      Running   0           2d
redis-slave-wwcxw   1/1      Running   0           2d

```

And then go into one of those pods and look at the database to see that everything looks right:

```

$ kubectl exec -it redis-slave-kd7vx redis-cli

```

```

127.0.0.1:6379> keys *
1) "guestbook"
127.0.0.1:6379> lrange guestbook 0 10
1) "hello world"
2) "welcome to the Kube workshop"
127.0.0.1:6379> exit

```



---

## Expose Redis Slave

Deploy redis slave service so we can access it by DNS name. Once redeployed, the application will send "read" operations to the `redis-slave` pods while "write" operations will go to the `redis-master` pods.

### `redis-slave-service.yaml`

```
apiVersion: v1
kind: Service
metadata:
  name: redis-slave
  labels:
    app: redis
    role: slave
spec:
  ports:
    - port: 6379
      targetPort: redis-server
  selector:
    app: redis
    role: slave
```

Create the service to access redis slaves. `$ kubectl create -f redis-slave-service.yaml`

Restart guestbook so that it will find the slave service to read from. `$ kubectl delete deploy guestbook-v1 $ kubectl create -f guestbook-deployment.yaml`

Test guestbook app using a browser of your choice using the url `<your-cluster-ip>:<node-port>`.

That's the end of the lab. Now let's clean-up our environment:

```
$ kubectl delete -f guestbook-deployment.yaml
kubectl delete -f guestbook-service.yaml
kubectl delete -f redis-slave-service.yaml
```

```
kubectrl delete -f redis-slave-deployment.yaml  
kubectrl delete -f redis-master-service.yaml  
kubectrl delete -f redis-master-deployment.yaml
```